COMP 210: Object-Oriented Programming **Lecture Notes 8**

Trees and Tree Traversals

Based on notes by Logan Mayfield

In these notes we look at Binary Trees and how to traverse them.

Binary Trees

Imagine a list. Now instead of every non-empty list having a single *next* list, what would happen if you allowed two nexts? What you'd have is a binary tree like the one seen in figure 1.

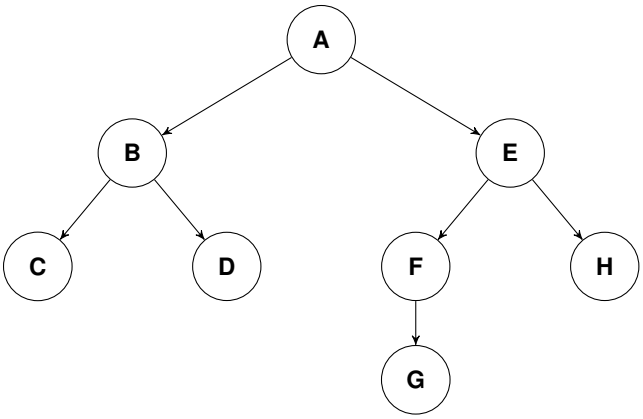

Figure 1: A Binary Tree

Great. Let's define this kind of structure using a PDA. If we keep the natural base case of an empty tree then we get a simple extension of a list.

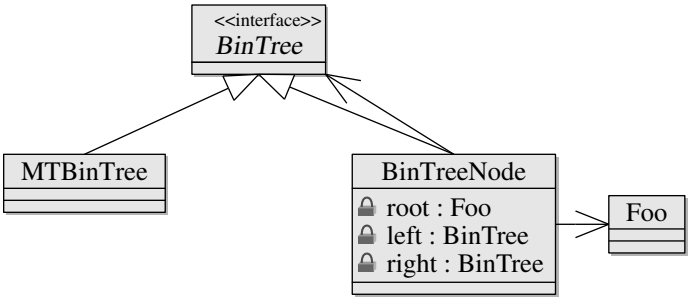

Figure 2: A Binary Tree of Foo Objects

If we wanted, we could then redraw our example tree to show the empty trees as shown in figure 3. This new diagarm gives us a pretty good feel for the actual structure of a BinTree object where the previous diagram[1] highlighted the logical structure of the tree.

[1] figure 1

It makes sense that the explosion of "nexts" leads to an explosion of empty trees. It's often convenient to work with another base case, the singleton tree. There's no reason we can't have two base cases, so let's add this variant of a binary tree to our previous design. The new PDA design given in figure 4 allows both empty an singleton base cases. It's not yet clear if this is helpful or just extra work, but just

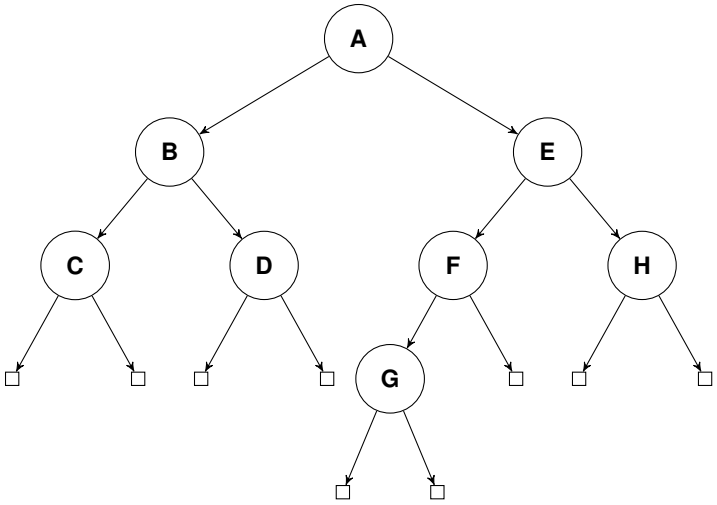

Figure 3: A Binary Tree with Empty Trees shown

drawing simple diagrams shows this might be a nice capability to have as the tree given in figure 5 both covers the complete structure and minimizes the need for a large number of empty trees.

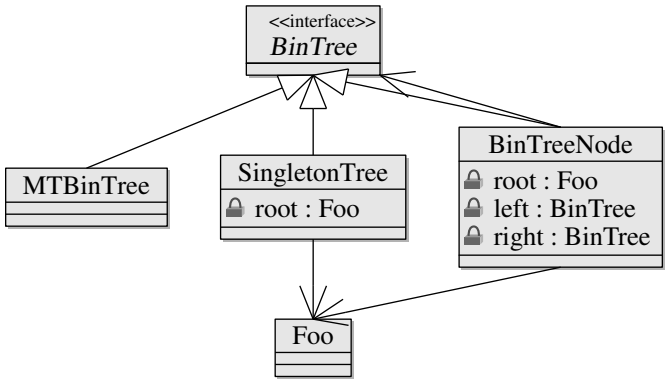

Figure 4: A Binary Tree of Foo Objects with Two Base Cases

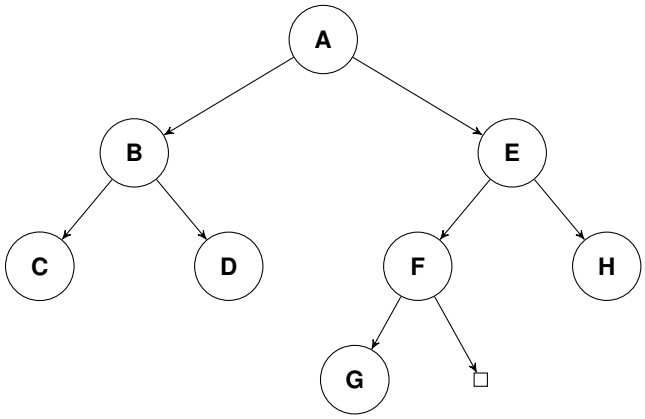

Figure 5: A Binary Tree with Empty Trees shown

At this point, it's worth imagining how the tree shown in figure 5 might be constructed in Java. For the sake of simplicity, we'll just assume the letters in each circle correspond to the name of some Foo variable.

```
BinTree atree =
  new BinTreeNode( A ,
    new BinTreeNode( B , new SingletonTree(C), new
        SingletonTree(D)),
    new BinTreeNode( E ,
      new BinTreeNode(F, new SingletonTree(G), new MTBinTree()),
       new SingletonTree(H))
    );
```

Figure 6: The tree in figure 5 as a Java BinTree

So we have some foothold into the world of trees now. We can define them as recursive class hierarchy and could easily start implementing functionality for that hierarchy so that we could play around with Binary Trees. Before we do that we really should first look at trees in a much broader context and find something interesting to do with trees.

Binary Tree Traversal

Traversing a list is fairly straight forward. From the recursive perspective you either recurse on the rest then deal with the first or your deal with the first then recurse on the rest. If we were computing the size of a list that might look like the snippets we see in figure 7.

```
// first, then rest
return 1 + this.rst.size();

// rest, then first
return this.rst.size() + 1;
```

Figure 7: Recursive List Traversal for *size*

To generalize these patterns we talk about what we do for the first as *visiting* the current node of the list. A little analysis shows that visiting then recursing will visit nodes in first to last order while recurse then visit will visit in last to first.

With trees we don't just have one recursively defined field but two: the left and right. What are we to do? The answer is simple, pick a permutation of visit (v), go left (l), and go right (r). There are six such permutations: vlr, lvr, lrv, vrl, rvl, and rlv. To simplify matters we only consider permutations where l comes before r. This leaves us with the three DEPTH FIRST traversal patterns.

1. PREORDER TRAVERSAL: v l r
2. INORDER TRAVERSAL: l v r
3. POSTORDER TRAVERSAL: l r v

The names are derived from the placement of the visit within the

permutation. In figure 8 we see how each of these patterns could be used to find the size[2] in a tree.

[2] number of non-empty nodes

```
// preorder
return 1 + this.left.size() + this.right.size();

// inorder
return this.left.size() + 1 + this.right.size();

//postorder
return this.left.size() + this.right.size() + 1;
```

Figure 8: Recursive Binary Tree Traversal for *size*

If you wanted to carry these processes out using an iterative loop then you'd need the assistance of a data structure that you studied in COMP220[3]. We'll save that task for another time. Right now it's more important to understand the order in which we'll visit a tree's nodes when doing each traversal. For our example tree from figure 5 we'd do the visit orders shown in table 1.

[3] a stack

Pattern	Visit Order
Preorder (vlr)	A B C D E F G H
Inorder (lvr)	C B D A G F E H
Postorder (lrv)	C D B G F H E A

Table 1: Traversal orders for the tree in figure 5

Doing these traversals by hand is wonderful practice for thinking through recursive a process. Just like some problems with lists necessitate a last to first traversal and some first to last, there will be tree problems that require different orders for visiting.

Talking Trees

Trees have been studied in mathematics for a long time and have found use in computing for as long as there has been computing. This means there is a lot of terminology to go along with trees. Much of it pulls from one of two metaphors, family trees or trees in nature.

- **Tree Structure**
  - Node A non-empty tree structure containing a single element and a left and right *subtree*.
  - Edge The connection between a Node and one of its subtrees.
  - Path A linearly connected set of nodes
    In our example tree, there is a path from the node containing A to the node containing G through the nodes E and F.

- ROOT NODE The Root node of a tree is the "first" node in that tree.
  A is the root of the tree shown in figure 5. It's left subtree is rooted by the node containing B and its right subtree is rooted by the node containing E.
- CHILD NODE The root node of a node's subtree.
  The node containing B is a child of the node containing A. Children of children and further down are called *descendents*. The node containing G is a descendent of the node containing E but the node containing D is not a descendent of the node containing E.
- PARENT NODE The node of which a given node is a child.
  The node containing A is the parent of the node containing B. Notice *the* root of a tree has no parent. Parents of parents and so on up the tree are called *ancestors*. The node containing E is an ancestor of the node containing G but not of the node containing D.
- LEAF NODE A node with no children
  These nodes are the bottom most layer of a tree. They're what we call the singleton tree and an alternate base case to the empty tree. The nodes containing C, D, G, and H are all leaf nodes. All other nodes are *internal* nodes because they have at least one child.

- **Properties of Trees and Nodes**
  - NODE DEPTH The number of edges from a given node to the tree's root node.
    The root of a tree always has a depth of 0 and from there depth increases by one with each level.
  - NODE HEIGHT The number of edges on the longest path from a give node to a leaf node.
  - TREE HEIGHT The height of the tree's root node.
    This property is the tree analog of list length.
  - TREE SIZE The number of nodes in the tree.
    Our example tree has a size of 8.

- **Classes of Binary Trees**
  - (HEIGHT) BALANCED BINARY TREE A tree in which the height of the left subtree and the height of the right subtree differ by at most one.
    Our example tree given in figure 5 is height balanced because the left subtree has a height of 1 and the right subtree has a height of 2.
  - COMPLETE BINARY TREE A tree of height h where the tree is full up to depth $h-1$ and then at depth h the leaves fill in from left to right.
    Complete trees are height balanced.

– FULL BINARY TREE A tree where every node except leaf nodes has exactly two children. // Full trees a special sub-class of Complete trees where every node at depth $h - 1$ has two children.

We've seen a height balanced tree already. Figure 9 shows you a complete tree of height 2 and figure 10 shows you a full tree of height 2.

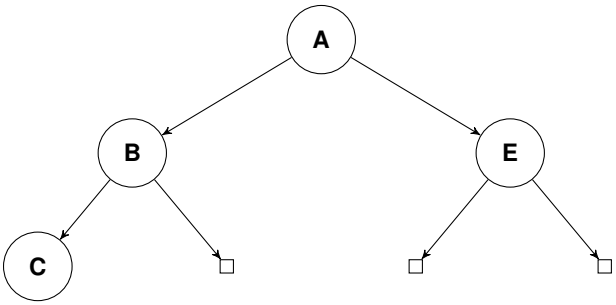

Figure 9: A Complete Binary Tree of height 2

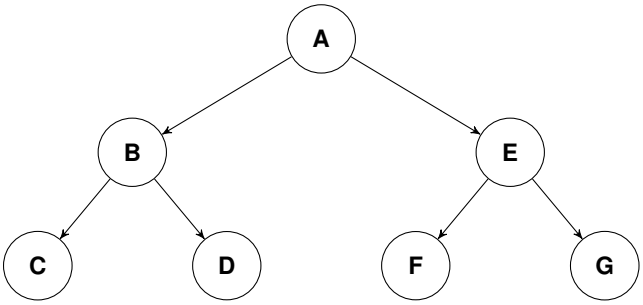

Figure 10: A Full Binary Tree of height 2

References